Typal Heterogeneous Equality Types

Andrew M.Pitts

University of Cambridge, UK

Abstract

The usual homogeneous form of equality type in Martin-Löf Type Theory contains identifications between elements of the same type. By contrast, the heterogeneous form of equality contains identifications between elements of possibly different types. This paper introduces a simple set of axioms for such types. The axioms are equivalent to the combination of systematic elimination rules for both forms of equality, albeit with typal (also known as "propositional") computation properties, together with Streicher's Axiom K, or equivalently, the principle of uniqueness of identity proofs.

1 Introduction

Equality types in the intensional version of Martin-Löf Type Theory (see for example Nordström et al., 1990, Section 8.1) are traditionally formulated in terms of an introduction rule (reflexivity) together with a rule for eliminating proofs of equality and a rule describing how elimination computes when it meets a reflexivity proof. Some recent work (Bezem et al., 2014; Cohen et al., 2018) on models of Homotopy Type Theory (Univalent Foundations Program, 2013) uses a formulation of equality types that differs from this in two respects. First, the elimination operation is replaced by the combination of a simple operation for transporting elements along proofs of equality, together with an axiom asserting contractibility of singleton types. Secondly, the analogue of the computation rule for the eliminator, namely that transporting along a reflexivity proof does nothing, is weakened from a judgemental equality to the existence of an element of the corresponding equality type; see Coquand (2011) and Figure 2 in (Bezem et al., 2014). This formulation is sometimes called a "propositional" equality type (van den Berg, 2018), but here I will follow Shulman (2018, Section 1.6) for the reasons given there and refer to *typal* equality types. Although these changes to the formulation of equality types affect computation, it seems that they do not change what is provable (see Boulier and Winterhalter (2019) for example) and they make it easier to construct models. Furthermore, they can lead to surprising simplifications. For example, Lumsdaine [private communication] has observed that the computation rule is superfluous (for elimination, but the observation also holds for transport): if a proto-identity type has a transport operation lacking its typal computation property, then the operation can be corrected to a new one that does have the computation property (see Lemma 2.1 and the Appendix).

Axioms for typal heterogeneous equality satisfying Axiom K

```
postulate
  _≡≡_ : ∀{l}{A B : Set l} → A → B → Set l

-- the derived homogeneous equality
_≡_  : ∀{l}{A : Set l} → A → A → Set l
x ≡ y = x ≡≡ y

postulate
  rfl : ∀{l}{A : Set l} (x : A) → x ≡ x
  ctr : ∀{l}{A B : Set l}{x : A}{y : B}(e : x ≡≡ y) → rfl x ≡≡ e
  eqt : ∀{l}{A B : Set l}{x : A}{y : B} → x ≡≡ y → A ≡ B
  tpt : ∀{l m n}{A : Set l}{B : A → Set m}(C : (x : A) → B x → Set n)
        {x x′ : A}{y : B x}{y′ : B x′} → x ≡ x′ → y ≡≡ y′ → C x y → C x′ y′
```

Axioms for Σ-types with surjective pairing

```
postulate
  ∑ : ∀{l m}(A : Set l)(B : A → Set m) → Set (l ⊔ m)

module _ {l m}{A : Set l}{B : A → Set m} where
  postulate
    _,_ : (x : A) → B x → ∑ A B
    fst : ∑ A B → A
    snd : (z : ∑ A B) → B (fst z)
    fpr : (x : A)(y : B x) → fst(x , y) ≡ x
    spr : (x : A)(y : B x) → snd(x , y) ≡≡ y
    eta : (z : ∑ A B) → (fst z , snd z) ≡ z

-- concrete syntax for ∑-types
syntax ∑ A (λ x → B) = ∑ x : A , B
```

Figure 1: The Axioms

The above remarks apply to the usual, homogeneous notion of equality in which elements of the same type are compared. The purpose of this paper is to give an analogous treatment of *heterogeneous* equality (McBride, 1999; Altenkirch et al., 2007) in the presence of Σ-types and the Axiom K of Streicher (1993, Section 1.2). Since Axiom K is not compatible with the Univalence Principle of Homotopy Type Theory (Univalent Foundations Program, 2013, Example 3.1.9), the focus here is on the simpler (but still useful!) world of zero-dimensional type theory. We will see that the axioms in Fig. 1 capture homogeneous and heterogeneous equality satisfying their usual dependent elimination and (typal) computation properties and Axiom K, and Σ-types with their usual dependent elimination and (typal) computation properties. It seems necessary to include Σ-types in order to get Lumsdaine's result mentioned above (see Remark 2.5); the axioms we give for such types are standard, except that the equality property of dependent second projection (`spr`) is simplified by the use of heterogeneous rather than homogeneous equality. The axioms in the figure are pleasingly simple compared to the usual formulation in terms of elimination and computation properties, and may aid finding new models of heterogeneous equality types.

The implementation of intensional Martin-Löf Type Theory provided by Agda 2.6 (Agda Wiki, [n.d.]) is used to state the axioms and develop their properties. More precisely, we just make use of Agda's implementation of a countably infinite, non-cumulative hierarchy of universes `Set l`, where `l` ranges over a type `Level` of universe levels whose closed normal forms are in bijection with the natural numbers. The universes are closed under dependent function types (written in Agda as $(x : A) \to B$) and inductive types. The use of a whole hierarchy of universes is necessary; for example, the function `eqt` in Fig. 1 takes a heterogeneous equality type $x \equiv\!\equiv y$ in universe `Set l` and produces a homogeneous one $A \equiv B$ in the universe one level up, which is denoted `Set(lsuc l)` in Agda. We also use Agda's notation for infix and for implicit arguments. For example, the function `_≡≡_` in Fig. 1 takes five arguments, the first three of which are implicit and the last two of which are infix. In particular, Agda's ability to infer the values of implicit arguments (or of unspecified explicit arguments, which are denoted by an underscore, `_`) is used quite aggressively in what follows, in order to be able to see the wood from the trees.

Although the code in this paper has been checked by Agda, some parts of it that are not essential for understanding the development have been hidden; the complete (non-literate) Agda code can be found at [`https://doi.org/10.17863/CAM.47902`].

2 The axioms and their properties

Figure 1 postulates a family of types `_≡≡_` in all universes, together with some operations on them that together capture a typal version of *heterogeneous* equality. Heterogeneous equality types were introduced by McBride (1999, Section 5.1.3) under the name of "John Major equality". Unlike ordinary, homogeneous equality types, such a type $x \equiv\!\equiv y$ relates elements x and y of possibly different types, A and B say. The intention is that elements of type $x \equiv\!\equiv y$ denote proofs that not only are x and y equal, but so also are their types A and B. The figure defines homogeneous equality `_≡_` as the special case of `_≡≡_` when the types of the two arguments are

already known to be the same. Axiom `rfl` says that ≡ is reflexive. Axiom `ctr` is a heterogeneous version of the contractibility property of singleton types (cf. center in Figure 2 of Bezem et al. (2014)). Axiom `eqt` says that heterogeneously equal things have (homogeneously) equal types. Axiom `tpt` is a form of the transport property of equality (cf. T in Figure 2 of Bezem et al. (2014)) involving both homogeneous and heterogeneous equalities. Finally, ∑, `_,_`, `fst`, `snd`, `fpr`, `spr` and `eta` axiomatize dependent product types satisfying surjective pairing.

We begin with some simple lemmas establishing the basics of equational logic for ≡≡, namely chain-reasoning using reflexivity (already an axiom), symmetry, transitivity and congruence properties. These are given in Fig. 2.

The axioms in Fig. 1 are notably lacking a "regularity" property for `tpt`, that is, a proof of type `tpt (rfl x) (rfl y) z ≡ z`. But such a thing is needed if we are to derive the expected elimination and (typal) computation rules for `_≡≡_` and `_≡_`. To get those, one can define a "corrected" form of transport that has this regularity property, using a simplified version of a trick due to Peter Lumsdaine [unpublished]. In fact, it is enough to produce a function coercing proofs of equality of types `e : A ≡ B` into functions `coe e : A → B` and which satisfies the heterogeneous regularity property that `coe e x ≡≡ x` (so that, given how we define ≡ in terms of ≡≡, the usual form of regularity, `coe (rfl A) x ≡ x`, is just the special case of this when `e` is `rfl A`).

Lemma 2.1. *The axioms in Fig. 1 imply the existence of a coercion function*

```
coe : ∀{l}{A B : Set l} → A ≡ B → A → B
```

satisfying a heterogeneous regularity property:

```
coeIsRegular : ∀{l}{A B : Set l}(e : A ≡ B)(x : A) → coe e x ≡≡ x
```

Proof. First we define the type of functions that are injective with respect to ≡ and note that the identity function is one such:

```
Inj     : ∀{l}(A B : Set l) → Set l
Inj A B = ∑ f : (A → B) , ∀{x y} → f x ≡ f y → x ≡ y

id      : ∀{l}{A : Set l} → A → A
id x    = x

idInj   : ∀{l}(A : Set l) → Inj A A
idInj _ = (id , id)
```

Next we use `tpt` to define a function coercing equalities into injective functions:

```
icoe : ∀{l}{A B : Set l} → A ≡ B → Inj A B
icoe {l} {A} e = tpt (λ _ C → Inj A C) (rfl (Set l)) e (idInj A)
```

The injectiveness of `icoe e` is used as follows. Applying the operation `tpt` to the type family

```
symm : ∀{l}{A B : Set l}{x : A}{y : B} → x ≡≡ y → y ≡≡ x
symm e = tpt (λ _ y → y ≡≡ _) (eqt e) e (rfl _)

proof_ : ∀{l}{A B : Set l}{x : A}{y : B} → x ≡≡ y → x ≡≡ y
proof p = p

_≡≡[_]_ : ∀{l}{A B C : Set l}(x : A){y : B}{z : C} →
          x ≡≡ y → y ≡≡ z → x ≡≡ z
x ≡≡[ e ] f = tpt (λ _ z → x ≡≡ z) (eqt f) f e

_qed : ∀{l}{A : Set l}(x : A) → x ≡ x
x qed = rfl x

cong : ∀{l m}{A : Set l}{B : A → Set m}(f : (x : A) → B x){x y : A} →
       x ≡ y → f x ≡≡ f y
cong f {x} e = tpt (λ _ z → f x ≡≡ f z) e e (rfl (f x))

cong₂ : ∀{l m n}{A : Set l}{B : A → Set m}{C : (x : A) → B x → Set n}
        (f : (x : A)(y : B x) → C x y){x x' : A}{y : B x}{y' : B x'} →
        x ≡ x' → y ≡≡ y' → f x y ≡≡ f x' y'
cong₂ f {x} {_} {y} e e' = tpt (λ x' y' → f x y ≡≡ f x' y') e e' (rfl (f x y))

cong₃ : ∀{k l m n}{A : Set k}{B : A → Set l}{C : (x : A) → B x → Set m}
        {D : (x : A)(y : B x) → C x y → Set n}
        (f : (x : A)(y : B x)(z : C x y) → D x y z)
        {x x' : A}{y : B x}{y' : B x'}{z : C x y}{z' : C x' y'} →
        x ≡ x' → y ≡≡ y' → z ≡≡ z' → f x y z ≡≡ f x' y' z'
cong₃ f {x} {_} {y} {_} {z} e e' =
  tpt (λ x' y' → ∀{z'} → z ≡≡ z' → f x y z ≡≡ f x' y' z') e e' (cong₂ (f x) (rfl y))
```

Figure 2: Equational reasoning for heterogeneous equality

```
fsticoe : ∀{l}{A : Set l}(x : A)(B : Set l)(e : A ≡ B) → Set l
fsticoe x B e = ∑ y : B , (fst (icoe (rfl B)) y ≡ fst (icoe e) x)
```

we can transport the element `(e , rfl (fst (icoe (rfl A)) x)` of type `fsticoe x A (rfl A)` along `e : A ≡ B` and `ctr e : rfl A ≡≡ e` to give an element of type `fsticoe x B e`. The first projection of this element gives the value of the desired coercion along `e` at `x`:

```
coe e x = fst (tpt (fsticoe x) e (ctr e) (x , rfl _))
```

and its second projection can be used along with the injectiveness property of `icoe` to get the regularity property of this coercion:

```
coeIsRegular {_} {A} e x = tpt (λ _ e' → coe e' x ≡≡ x) e (ctr e) coerfl
  where
  coerfl : coe (rfl A) x ≡ x
  coerfl = snd (icoe (rfl A)) (snd (
             tpt (fsticoe x) (rfl A) (ctr (rfl A)) (x , rfl _)))
```

□

An immediate corollary is that the axioms imply the *uniqueness of identity proofs* (UIP) and hence Streicher's Axiom K (Streicher, 1993). (We will see in Sect. 3 that in fact it is only the `tpt` function that contains an implicit use of Axiom K.)

Theorem 2.2 (UIP and Axiom K)**.** *The axioms in Fig. 1 imply that* ≡ *satisfies*

```
uip        : ∀{l}{A : Set l}{x y : A}(e e' : x ≡ y) → e ≡ e'

axiomK     : ∀{l m}{A : Set l}{x : A}(P : x ≡ x → Set m)(p : P (rfl x)) →
             ∀ e → P e

axiomKComp : ∀{l m}{A : Set l}{x : A}(P : x ≡ x → Set m)(p : P (rfl x)) →
             axiomK P p (rfl x) ≡ p
```

Proof. Using the functions from Fig 2 and Lemma 2.1 we have:

```
uip e e'          = tpt (λ _ e'' → e'' ≡≡ e') e (ctr e) (ctr e')

axiomK P p e      = coe (cong₂ (λ _ → P) (rfl p) (ctr e)) p

axiomKComp P p = proof
                   coe (cong₂ (λ _ → P) (rfl p) (ctr (rfl _))) p
                 ≡≡[ cong (λ e → coe e p) (uip _ _) ]
                   coe (rfl _) p
                 ≡≡[ coeIsRegular _ p ]
                   p
                 qed
```

□

The elimination and computation properties of $\equiv$ and $\equiv\equiv$ then follow:

Theorem 2.3 (Elimination and typal computation properties)**.** *The axioms in Fig. 1 imply that* $\equiv$ *has the usual elimination and (typal) computation properties of homogeneous equality (in the form suggested by Paulin-Mohring (1993))*

```
≡Elim : ∀{l m}{A : Set l}{x : A}(P : (y : A) → x ≡ y → Set m)
        (p : P x (rfl x))(y : A)(e : x ≡ y) → P y e
≡Comp : ∀{l m}{A : Set l}{x : A}(P : (y : A) → x ≡ y → Set m)
        (p : P x (rfl x)) → ≡Elim P p x (rfl x) ≡ p
```

The axioms also imply that $\equiv\equiv$ *has the elimination and (typal) computation properties of heterogeneous equality described by McBride (1999, Section 5.1.3)*

```
≡≡Elim : ∀{l m}{A : Set l}{x : A}(P : (B : Set l)(y : B) → x ≡≡ y → Set m)
         (p : P A x (rfl x))(B : Set l)(y : B)(e : x ≡≡ y) → P B y e

≡≡Comp : ∀{l m}{A : Set l}{x : A}(P : (B : Set l)(y : B) → x ≡≡ y → Set m)
         (p : P A x (rfl x)) → ≡≡Elim P p A x (rfl x) ≡ p
```

Proof. Using the functions from Fig 2 and Lemma 2.1 we have:

```
≡Elim P p _ e    = coe (cong₂ P e (ctr e)) p

≡Comp P p        = proof
                     coe (cong₂ P (rfl _) (ctr (rfl _))) p
                   ≡≡[ cong (λ e → coe e p) (symm (ctr _)) ]
                     coe (rfl _) p
                   ≡≡[ coeIsRegular _ p ]
                     p
                   qed

≡≡Elim P p _ _ e = coe (cong₃ P (eqt e) e (ctr e)) p

≡≡Comp P p       = proof
                     coe (cong₃ P (eqt (rfl _)) (rfl _) (ctr (rfl _))) p
                   ≡≡[ cong (λ e → coe e p) (uip _ _) ]
                     coe (rfl _) p
                   ≡≡[ coeIsRegular _ p ]
                     p
                   qed
```

□

Note that a corollary of the above two theorems is that _ ≡≡ _ is uniquely determined up to logical equivalence by the axioms in Fig. 1. In other words, for any other such family of types _ ≡≡′ _, there are functions in either direction between $x \equiv\equiv y$ and $x \equiv\equiv' y$; and because of UIP these are necessarily mutually inverse up to ≡≡ (or ≡≡′).

Remark 2.4. ≡≡Elim is the elimination form systematically derived (Backhouse et al., 1989) from _ ≡≡ _ and rfl, regarding them as the formation and introduction rules for an inductive type. As McBride (1999, page 120) points out, ≡≡Elim is not very useful, because of the way it's motive P involves abstraction over an arbitrary type B. McBride goes on to give another, more useful form of elimination for ≡≡, but in our setting where ≡ is a special case of ≡≡, that coincides with the eliminator ≡Elim.

Remark 2.5 (The role of Σ-types)**.** One of the strengths of machine-checked mathematics is that it aids the detection of logical dependency. Although we included the equations fpr, spr and eta for Σ-types in Fig. 1, they have not been used for the results so far, as may be verified by commenting them out from this literate Agda file and re-checking it up to this point.

So only the weak form of dependent product given by ∑, _,_, fst and snd in the figure is used to define the regular version of coercion in Lemma 2.1 and then prove Theorems 2.2 and 2.3. It would be nice if there was some way to define ∑, _,_, fst and snd just using dependent function types and universes.

However, the extra equations fpr, spr and eta for ∑ are of course very natural. Let us record the fact that they enable one to define the usual elimination rule for dependent products, with a typal computation rule:

```
∑Elim      : ∀{l m n}{A : Set l}{B : A → Set m}(C : ∑ A B → Set n)
             (c : (x : A)(y : B x) → C (x , y))(z : ∑ A B) → C z
∑Elim C c z = coe (cong C (eta z)) (c (fst z) (snd z))

∑Comp      : ∀ {l m n}{A : Set l}{B : A → Set m}(C : ∑ A B → Set n)
             (c : (x : A)(y : B x) → C (x , y))(x : A)(y : B x) →
             ∑Elim C c (x , y) ≡ c x y
∑Comp C c x y = let z = (x , y) in
                proof
                  coe (cong C (eta z)) (c (fst z) (snd z))
                ≡≡[ coeIsRegular _ _ ]
                  c (fst z) (snd z)
                ≡≡[ cong₂ c (fpr x y) (spr x y) ]
                  c x y
                qed
```

3 Consistency of the axioms

We have seen that the axioms in Fig. 1 suffice to define dependent products and both heterogeneous and homogeneous equality types with uniqueness of identity proofs, all satisfying the

usual elimination properties, albeit with typal computation rules. Conversely it is not hard to see that the elimination and computation rules in Theorem 2.3 and Remark 2.5, together with Axiom K, imply the axioms in Fig. 1. Instead of doing that, in this section we just check that the axioms are provable from inductive definitions of equality and dependent product types. One can make these inductive definitions in Agda as follows:

```
data _≡≡_ {l}{A : Set l} : {B : Set l} → A → B → Set l where
  rfl : (x : A) → x ≡≡ x
data ∑ {l m}(A : Set l)(B : A → Set m) : Set (l ⊔ m) where
  _,_ : (x : A) → B x → ∑ A B
-- the derived homogeneous equality
_≡_ : ∀{l}{A : Set l} → A → A → Set l
x ≡ y = x ≡≡ y
```

Then Agda's implementation of dependent pattern matching enables straightforward definitions of the functions from Fig. 1, as follows:

```
ctr : ∀{l}{A B : Set l}{x : A}{y : B}(e : x ≡≡ y) → rfl x ≡≡ e
ctr (rfl x) = rfl (rfl x)

eqt : ∀{l}{A B : Set l}{x : A}{y : B} → x ≡≡ y → A ≡ B
eqt {_} {A} (rfl _) = rfl A

tpt : ∀{l m n}{A : Set l}{B : A → Set m}(C : (x : A) → B x → Set n)
      {x x' : A}{y : B x}{y' : B x'} → x ≡ x' → y ≡≡ y' → C x y → C x' y'
tpt _ (rfl _) (rfl _) y = y

module _ {l m}{A : Set l}{B : A → Set m} where
  fst : ∑ A B → A
  fst (x , _) = x

  snd : (z : ∑ A B) → B (fst z)
  snd (_ , y) = y

  fpr : (x : A)(y : B x) → fst (x , y) ≡ x
  fpr x _ = rfl x

  spr : (x : A)(y : B x) → snd (x , y) ≡≡ y
  spr _ y = rfl y

  eta : (z : ∑ A B) → (fst z , snd z) ≡ z
  eta (x , y) = rfl (x , y)
```

Since we know from the previous section that these functions entail Axiom K, the above definitions have to use Agda's default `--with-K` option to switch the existing implementation of

dependent pattern matching (Cockx and Abel, 2018) back to the original version due to Coquand (1992), which is known to imply Axiom K (Goguen et al., 2006). More precisely, it is only the matches on the two occurrences of the pattern `rfl _` in the definition of `tpt` that involve an implicit use of Axiom K (to discharge the unification constraints $A \doteq A$ and $B\,x \doteq B\,x$); all the other functions can be defined without Axiom K.

4 Conclusion

This paper has investigated heterogeneous equality and produced a simple collection of axioms for its typal form, in the spirit of Coquand (2011). The point of view is foundational. From a practical perspective, the use of heterogeneous equality has much to recommend it for formalizing mathematics in dependent type theory when assuming uniqueness of identity proofs[1]; but that is another story.

References

Agda Wiki. [n.d.]. https://wiki.portal.chalmers.se/agda/pmwiki.php.

T. Altenkirch, C. McBride, and W. Swierstra. 2007. Observational equality, now!. In *PLPV '07: Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/1292597.1292608

R. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. 1989. Do-it-Yourself Type Theory. *Formal Aspects of Computing* 1 (1989), 19–84.

M. Bezem, T. Coquand, and S. Huber. 2014. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013) (Leibniz International Proceedings in Informatics (LIPIcs))*, R. Matthes and A. Schubert (Eds.), Vol. 26. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 107–128. https://doi.org/10.4230/LIPIcs.TYPES.2013.107

S. Boulier and T. Winterhalter. 2019. Weak Type Theory is Rather Strong. (June 2019). Abstract for the 25th International Conference on Types for Proofs and Programs (TYPES 2019), Oslo, Norway.

J. Cockx and A. Abel. 2018. Elaborating Dependent (Co)Pattern Matching. *Proc. ACM Program. Lang.* 2, ICFP, Article 75 (July 2018), 30 pages. https://doi.org/10.1145/3236770

C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, T. Uustalu (Ed.), Vol. 69. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:34. https://doi.org/10.4230/LIPIcs.TYPES.2015.5

[1] Such is the approach of Lean (The Lean Theorem Prover, [n.d.]) since version 3, for example.

T. Coquand. 1992. Pattern Matching with Dependent Types. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, B. Nordström, K. Petersson, and G. D. Plotkin (Eds.). 66–79.

T. Coquand. 2011. Equality and Dependent Type Theory. (Feb. 2011). A talk given for the 24th AILA meeting, Bologna (`http://www.cse.chalmers.se/~coquand/bologna.pdf`).

H. Goguen, C. McBride, and J. McKinna. 2006. Eliminating Dependent Pattern Matching. In *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, K. Futatsugi, J.-P. Jouannaud, and J. Meseguer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540. `https://doi.org/10.1007/11780274_27`

C. McBride. 1999. *Dependently Typed Functional Programs and their Proofs*. Ph.D. Dissertation. University of Edinburgh.

B. Nordström, K. Petersson, and J. M. Smith. 1990. *Programming in Martin-Löf's Type Theory*. Oxford University Press.

Chr. Paulin-Mohring. 1993. Inductive definitions in the system Coq; rules and properties. In *Typed Lambda Calculus and Applications (Lecture Notes in Computer Science)*, M. Bezem and J. F. Groote (Eds.), Vol. 664. Springer-Verlag, Berlin, 328–345.

M. Shulman. 2018. Brouwer's Fixed-Point Theorem in Real-Cohesive Homotopy Type Theory. *Mathematical Structures in Computer Science* 28 (2018), 856–941.

T. Streicher. 1993. *Investigations into Intensional Type Theory*. Habilitation Thesis. Ludwig Maximilian University, Munich.

The Lean Theorem Prover. [n.d.]. `https://leanprover.github.io`.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations for Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study.

B. van den Berg. 2018. Path Categories and Propositional Identity Types. *ACM Trans. Comput. Logic* 19, 2 (June 2018), 15:1–15:32. `https://doi.org/10.1145/3204492`

Appendix: typal homogeneous equality without K

In this appendix, for completeness sake we consider axioms in dependent type theory without Axiom K for *homogeneous* equality types

```
postulate
  _≡_ : ∀{ℓ}{A : Set ℓ} → A → A → Set ℓ
```

following Coquand (2011). (Since without Axiom K heterogeneous equality is not very useful, we do not bother to consider axiomatizing ≡≡ in that setting.) One of the axioms makes use

of dependent product types. Although one could axiomatize those types as we did in the main part of the paper, it is simpler to use an inductive defintion and corresponding pair patterns:

```
data ∑ {l m}(A : Set l)(B : A → Set m) : Set (l ⊔ m) where
  _,_ : (x : A) → B x → ∑ A B
-- concrete syntax for ∑-types
syntax ∑ A (λ x → B) = ∑ x : A , B
-- dependent product projections
module _ {l m}{A : Set l}{B : A → Set m} where
  fst : ∑ A B → A
  fst (x , _) = x
  snd : (z : ∑ A B) → B (fst z)
  snd (_ , y) = y
```

The axioms for homogeneous equality are

```
postulate
  refl : ∀{l}{A : Set l} (x : A) → x ≡ x
  cntr : ∀{l}{A : Set l}{x y : A}(e : x ≡ y) → (x , refl x) ≡ (y , e)
  sbst : ∀{l m}{A : Set l}(B : A → Set m){x x' : A} → x ≡ x' → B x → B x'
```

Coquand also considers a regularity axiom for `sbst` (ax_3 in *loc.cit.*), but one can do without that by using Peter Lumsdaine's trick to correct `sbst` to a version `subst` for which there is a proof `substIsRegular : ∀b → subst (refl x) b ≡ b`, as follows. The proof begins as for Lemma 2.1 by considering functions that are injective modulo ≡:

```
Inj : ∀{l}(A B : Set l) → Set l
Inj A B = ∑ f : (A → B) , ∀{x y} → f x ≡ f y → x ≡ y

id : ∀{l}{A : Set l} → A → A
id x = x

idInj : ∀{l}(A : Set l) → Inj A A
idInj _ = (id , id)
```

But then to construct `subst` and `substIsRegular`, one has to work a bit harder than in the proof of the lemma, because of the lack of uniqueness of identity proofs:

```
module _ {l m}{A : Set l}(B : A → Set m){x : A} where
  Inj₂ : {y z : A} → x ≡ y → x ≡ z → Inj (B y) (B z)
  Inj₂ {y} p q =
    sbst (λ z' → Inj (B y) (B z')) q
      (sbst (λ y' → Inj (B y') (B x)) p (idInj (B x)))
```

```
sbst₂ : {y z : A} → x ≡ y → x ≡ z → B y → B z
sbst₂ p q = fst (Inj₂ p q)

C : {y : A}(p : x ≡ y)(b : B x) → ∑ c : B y , (sbst₂ p p c ≡ sbst₂ (refl x) p b)
C p b = sbst C′ (cntr p) (b , refl _)
  where
  C′ : ∑ y : A , (x ≡ y) → Set m
  C′ (y , p) = ∑ c : B y , (sbst₂ p p c ≡ sbst₂ (refl x) p b)

subst : {y : A} → x ≡ y → B x → B y
subst p b = fst (C p b)

substIsRegular : (b : B x) → subst (refl x) b ≡ b
substIsRegular b = snd (Inj₂ (refl x) (refl x)) (snd (C (refl x) b))
```